The graph rewriting calculus: confluence and expressiveness

Clara Bertolissi

The graph rewriting calculus: confluence and expressiveness

Clara Bertolissi

LORIA & UHP, BP 239 54506 Vandoeuvre-lès-Nancy Cedex France
clara.bertolissi@loria.fr

Abstract. Introduced at the end of the nineties, the Rewriting Calculus (ρ-calculus, for short) is a simple calculus that uniformly integrates term-rewriting and λ-calculus. The $\rho_{\mathtt{g}}$-calculus has been recently introduced as an extension of the ρ-calculus, handling structures with cycles and sharing. The calculus over terms is naturally generalized by using unification constraints in addition to the standard ρ-calculus matching constraints. This leads to a term-graph representation in an equational style where terms consist of unordered lists of equations. In this paper we show that the (linear) $\rho_{\mathtt{g}}$-calculus is confluent. The proof of this result is quite elaborated, due to the non-termination of the system and to the fact that we work on equivalence classes of terms. We also show that the $\rho_{\mathtt{g}}$-calculus can be seen as a generalization of first-order term-graph rewriting, in the sense that for any term-graph rewrite step a corresponding sequence of rewritings can be found in the $\rho_{\mathtt{g}}$-calculus.

1 Introduction

Term rewriting is a general framework for specifying and reasoning about computations that combines elements of automated theorem proving, universal algebra and functional programming. It provides very efficient methods for reasoning with equations and it can be regarded as a powerful abstract computational model. In particular, term rewriting systems can be used for software verification: the behavior of a functional or rewrite-based program can be described by analyzing some properties of the associated term rewriting system. For example, the confluence property ensures that the output of the program, if it exist, is unique for any given input data.

In this framework, the rewriting calculus (ρ-calculus, for short) has been introduced in the late nineties as a natural generalization of term rewriting and of the λ-calculus [9]. The rewrite rules, acting as elaborated abstractions, their application and the obtained structured results are first-class objects of the calculus. The evaluation mechanism, generalizing beta-reduction, strongly relies on term matching in various theories. Several variants of the calculus have been already studied, such as typed versions [5], extensions with explicit substitutions [8] or with imperative features [15].

In the term rewriting setting, terms are often seen as trees but in order to improve the efficiency of the implementation of functional languages, it is of

fundamental interest to think and implement terms as graphs [4]. In this case, the possibility of sharing subterms allows one to save space and time during the computation. Moreover, the possibility to define cycles leads to an increased expressive power that allows one to represent naturally regular infinite data structures. Cyclic term-graph rewriting has been widely studied from different points of view: operational [4], categorical [11] or equational [1] (see [18] for a survey on term-graph rewriting).

Following the last approach, we proposed in [7] a system, called ρ_{g}-calculus, that generalizes the standard ρ-calculus in order to deal with higher-order cyclic terms. The first contribution of this paper shows that the ρ_{g}-calculus, under some linearity assumptions, is confluent. In the ρ_{g}-calculus terms can have an associated list of constraints which is composed of recursion equations, used to express sharing and cycles, and matching constraints, arising from the fact that computations related to the matching are made explicit and performed at the object-level. The order of constraints is irrelevant and therefore the conjunction operator is considered to be commutative and associative, with the the empty constraint as neutral element. Moreover, the idempotence axiom is used to avoid the duplication of constraints.

The fact that reductions take place over equivalence classes of terms rather than over single terms must be considered when proving the confluence of the calculus, and this makes the result more difficult to achieve. The proof method generalizes the proof of confluence of the cyclic λ-calculus [2] to the setting of rewriting modulo an equational theory [16] and moreover it adapts the proof to deal with terms containing patterns and matching equations. More precisely, the proof, which is only sketched in the paper, uses the concept of "developments" and the property of "finiteness of developments" as defined in the theory of classical λ-calculus [3]. The interested reader can find the complete proof in [6].

The ρ_{g}-calculus is an expressive formalism that has already been shown to be a generalization of both the plain ρ-calculus and the λ-calculus extended with explicit recursion, providing an homogeneous framework for pattern matching and higher-order graphical structures. In this paper we show how the ρ_{g}-calculus can be naturally seen also as an extension of term-graph rewriting. More specifically, we prove that matching in the ρ_{g}-calculus is well-behaved *w.r.t.* the notion of homomorphism on term-graphs and that any reduction step in a term-graph rewrite system can be simulated in the ρ_{g}-calculus.

The paper is organized as follows. In Section 2 we describe the syntax and the small-step semantics of the ρ_{g}-calculus. In Section 3 we outline the proof of confluence for the ρ_{g}-calculus. In Section 4 we review first-order term-graph rewriting in the equational approach, showing that term-graph reductions can be simulated in the ρ_{g}-calculus. We conclude in Section 5 by presenting some perspectives of this work.

Terms

$$
\begin{array}{lll}
\mathcal{G}, \mathcal{P} ::= \mathcal{X} & & \text{(Variables)} \\
\quad | \ \mathcal{K} & & \text{(Constants)} \\
\quad | \ \mathcal{P} \twoheadrightarrow \mathcal{G} & & \text{(Abstraction)} \\
\quad | \ \mathcal{G}\ \mathcal{G} & & \text{(Functional application)} \\
\quad | \ \mathcal{G}; \mathcal{G} & & \text{(Structure)} \\
\quad | \ \mathcal{G}\ [\mathcal{C}] & & \text{(Constraint application)}
\end{array}
$$

Constraints

$$
\begin{array}{lll}
\mathcal{C} ::= \epsilon & & \text{(Empty constraint)} \\
\quad | \ \mathcal{X} = \mathcal{G} & & \text{(Recursion equation)} \\
\quad | \ \mathcal{P} \ll \mathcal{G} & & \text{(Match equation)} \\
\quad | \ \mathcal{C}, \mathcal{C} & & \text{(Conjunction)}
\end{array}
$$

Fig. 1. Syntax of the ρ_{g}-calculus

2 The ρ_{g}-calculus: syntax and semantics

The syntax of the ρ_{g}-calculus presented in Fig. 1 extends the syntax of the standard ρ-calculus and of the ρ_{x}-calculus [8], *i.e.* the ρ-calculus with explicit matching and substitution application. As in the plain ρ-calculus, λ-abstraction is generalized by a rule abstraction $P \twoheadrightarrow G$, where P is in general an arbitrary term. There are two different application operators: the functional application operator, denoted simply by concatenation, and the constraint application operator, denoted by as "$\_\ [\_]$". Terms can be grouped together into *structures* built using the operator "$\_; \_$". Depending on the theory behind this operator a structure can represent, e.g., a multi-set (when ";" is associative and commutative) or a set (when ";" is associative, commutative and idempotent) of terms.

In the ρ_{g}-calculus constraints are conjunctions (built using the operator "$\_, \_$") of match equations of the form $\mathcal{P} \ll \mathcal{G}$ and recursion equations of the form $\mathcal{X} = \mathcal{G}$. The empty constraint is denoted by ϵ. The operator "$\_, \_$" is supposed to be associative, commutative and idempotent, with ϵ as neutral element.

We assume that the application operator associates to the left, while the other operators associate to the right. To simplify the syntax, operators have different priorities. Here are the operators ordered from higher to lower priority: "$\_\ \_$", "$\_ \twoheadrightarrow \_$", "$\_; \_$", "$\_\ [\_]$" , "$\_ \ll \_$", "$\_ = \_$" and "$\_, \_$".

The symbols $G, H, P \ldots$ range over the set $\mathcal{G}$ of terms, $x, y, \ldots$ range over the set $\mathcal{X}$ of variables, $a, b, \ldots$ range over a set $\mathcal{K}$ of constants. The symbols $E, F, \ldots$ range over the set $\mathcal{C}$ of constraints. We call *algebraic* the terms of the form $(((f\ G_1)\ G_2) \ldots)\ G_n$, with $f \in \mathcal{K}$, $G_i \in \mathcal{X} \cup \mathcal{K}$ or G_i algebraic for $i = 1 \ldots n$, and we usually denote them by $f(G_1, G_2, \ldots, G_n)$.

We denote by $\bullet$ (black hole) a constant, already introduced in [1] using the equational approach and also in [11] using the categorical approach, to give a name to "undefined" terms that correspond to the expression $x\ [x = x]$ (self-loop). The notation $x =_\circ x$ is an abbreviation for the sequence $x = x_1, \ldots, x_n = x$. We use the symbol $\mathsf{Ctx}\{\Box\}$ for a context with exactly one hole $\Box$. We say that a ρ_{g}-term is *acyclic* if it contains no sequence of constraints of the form $\mathsf{Ctx}_0\{x_0\} \lll \mathsf{Ctx}_1\{x_1\}, \mathsf{Ctx}_2\{x_1\} \lll \mathsf{Ctx}_3\{x_2\}, \ldots, \mathsf{Ctx}_m\{x_n\} \lll \mathsf{Ctx}_{m+1}\{x_0\}$, with $n, m \in \mathbb{N}$ and $\lll \in \{=, \ll\}$. A sequence of this kind is called a *cycle*.

For the purpose of this paper we restrict to left-hand sides of abstractions and match equations that are acyclic, algebraic terms, with all their subterms algebraic and not containing constraints. The set of all these terms, called *patterns*, is denoted by $\mathcal{P}$. For instance, the ρ_{g}-term $(f(y)\ [y = g(y)] \twoheadrightarrow a)$ is not allowed since the abstraction has a cyclic left-hand side. We call a ρ_{g}-term *well-formed* if each variable occurs at most once as left-hand side of a recursion equation. All the ρ_{g}-terms considered in the sequel will be implicitly well-formed.

The notions of free and bound variables of ρ_{g}-terms take into account the three binders of the calculus: abstraction, recursion and match. Intuitively, variables on the left hand-side of any of these operators are bound by the operator. The set of free variables of a ρ_{g}-term G is denoted by $\mathcal{FV}(G)$. Moreover, given a constraint $\mathcal{C}$ we will refer to the set $\mathcal{DV}(\mathcal{C})$, of variables "defined" in $\mathcal{C}$. This set includes, for any recursion equation $x = G$ in $\mathcal{C}$, the variable x and for any match $P \ll G$ in $\mathcal{C}$, the set of free variables of P. For a formal definition, see [7].

We work modulo α-*conversion* and we use Barendregt's *"hygiene-convention"*, *i.e.* free and bound variables have different names [3]. Note that the scope of a recursion variable is limited to the ρ_{g}-terms appearing in the list of constraints where such variable is defined and the ρ_{g}-term to which this list is applied. For example, in $f(x, y)\ [x = g(y)\ [y = a]]$ the variable y defined in the recursion equation binds its occurrence in $g(y)$ but not in $f(x, y)$. In fact, the term does not satisfy the naming conditions since y occurs both free and bound. This naming convention allows us to apply replacements (like for the evaluation rules in Fig. 2) quite straightforwardly, since no variable capture is possible.

We define next an order over variables bound by a match or an equation. This order will be later used in the definition of the substitution rule of the calculus, which will allow one only upward substitutions. As we will see later, this is essential for obtaining the confluence of the calculus. We denote by $\leq$ the least pre-order on recursion variables such that $x \geq y$ if $x = \mathsf{Ctx}\{y\}$, for some context $\mathsf{Ctx}\{\Box\}$. The equivalence induced by the pre-order is denoted $\equiv$ and we say that x and y are cyclically equivalent ($x \equiv y$) if $x \geq y \geq x$ (they lie on a common cycle). We write $x > y$ if $x \geq y$ and $x \not\equiv y$.

Example 1 (Some ρ_{g}-terms).

1. In the rule $(2 * f(x)) \twoheadrightarrow ((y + y)\ [y = f(x)])$ the sharing in the right-hand side avoids the copying of the object instantiating $f(x)$, when the rule is applied to a ρ_{g}-term.
2. The ρ_{g}-term $x\ [x = cons(0, x)]$ represents an infinite list of zeros.
3. The ρ_{g}-term $f(x, y)\ [x = g(y), y = g(x)]$ is an example of twisted sharing that can be expressed using mutually recursive constraints (to be read as a `letrec` construct). We have that $x \geq y$ and $y \geq x$, hence $x \equiv y$.

The complete set of evaluation rules of the ρ_{g}-calculus is presented in Fig. 2. As in the plain ρ-calculus, in the ρ_{g}-calculus the application of a rewrite rule to a term is represented as the application of an abstraction. A redex can be activated using the ρ rule in the BASIC RULES, which creates the corresponding matching constraint. The computation of the substitution which solves the matching is

BASIC RULES:

$$
\begin{array}{lll}
(\rho) & (P \twoheadrightarrow G_2)\ G_3 & \to_\rho\ G_2\ [P \ll G_3] \\
 & (P \twoheadrightarrow G_2)\ [E]\ G_3 & \to_\rho\ G_2\ [P \ll G_3, E] \\
(\delta) & (G_1; G_2)\ G_3 & \to_\delta\ G_1\ G_3; G_2\ G_3 \\
 & (G_1; G_2)\ [E]\ G_3 & \to_\delta\ (G_1\ G_3; G_2\ G_3)\ [E]
\end{array}
$$

MATCHING RULES:

$$
\begin{array}{lll}
(\mathit{propagate}) & P \ll (G\ [E]) & \to_p\ P \ll G, E \quad \mathit{if}\ P \neq x \\
(\mathit{decompose}) & K(G_1, \ldots, G_n) \ll K(G'_1, \ldots, G'_n) & \to_{dk}\ G_1 \ll G'_1, \ldots, G_n \ll G'_n \\
 & & \quad \mathit{with}\ n \geq 0 \\
(\mathit{solved}) & x \ll G, E & \to_s\ x = G, E \quad \mathit{if}\ x \notin \mathcal{DV}(E)
\end{array}
$$

GRAPH RULES:

$$
\begin{array}{lll}
(\mathit{external\ sub}) & \mathsf{Ctx}\{y\}\ [y = G, E] & \to_{es}\ \mathsf{Ctx}\{G\}\ [y = G, E] \\
(\mathit{acyclic\ sub}) & G\ [P \lll \mathsf{Ctx}\{y\}, y = G_1, E] & \to_{ac}\ G\ [P \lll \mathsf{Ctx}\{G_1\}, y = G_1, E] \\
 & & \quad \mathit{if}\ x > y,\ \forall x \in \mathcal{FV}(P) \\
 & & \quad \mathit{where}\ \lll \in \{=, \ll\} \\
(\mathit{garbage}) & G\ [E, x = G'] & \to_{gc}\ G\ [E] \\
 & & \quad \mathit{if}\ x \notin \mathcal{FV}(E) \cup \mathcal{FV}(G) \\
 & G\ [\epsilon] & \to_{gc}\ G \\
(\mathit{black\ hole}) & \mathsf{Ctx}\{x\}\ [x =_\circ x, E] & \to_{bh}\ \mathsf{Ctx}\{\bullet\}\ [x =_\circ x, E] \\
 & G\ [P \lll \mathsf{Ctx}\{y\}, y =_\circ y, E] & \to_{bh}\ G\ [P \lll \mathsf{Ctx}\{\bullet\}, y =_\circ y, E] \\
 & & \quad \mathit{if}\ \ x > y,\ \forall x \in \mathcal{FV}(P)
\end{array}
$$

Fig. 2. Small-step semantics of the ρ_{g}-calculus

then performed explicitly by the MATCHING RULES and, if the computation is successful, the result is a recursion equation added to the list of constraints of the term. This means that the substitution is not applied immediately to the term but it is kept in the environment for a delayed application or for deletion if useless, as expressed by the GRAPH RULES.

More precisely, the first two rules ρ and δ come from the ρ-calculus. The rule δ distributes the application over the the structures built with the ";" operator. The rule ρ triggers the application of a rewrite rule to a ρ_{g}-term by applying the appropriate constraint to the right-hand side of the rule. For each of these rules, an additional rule dealing with the presence of constraints is considered.

The MATCHING RULES and in particular the rule *decompose* are strongly related to the theory modulo which we want to compute the solutions of the matching. In this paper we consider the syntactical matching, which is known to be decidable, but extensions to more elaborated theories are possible. Due to the assumptions on the left-hand sides of rewrite rules and of constraints, we only need to decompose algebraic terms. The goal of this set of rules is to produce a constraint of the form $x_1 = G_1, \ldots, x_n = G_n$ starting from a matching equation. Some replacements might be needed (as defined by the GRAPH RULES) as soon as the terms contain some sharing. The *propagate* rule performs a flattening of a list of constraints which are propagated to the top level. The rule *solved* transforms a matching constraint $x \ll G$ into a recursion equation $x = G$. The proviso asking that x is not defined elsewhere in the constraint is necessary in

the case of matching problems involving non-linear constraints. For example, the constraint $x \ll a, x \ll b$ should not be reduced showing that the original (non-linear) matching problem has no solution.

The GRAPH RULES are inherited from the cyclic λ-calculus [2]. The first two rules make a copy of a ρ_{g}-term associated to a recursion variable into a term that is inside the scope of the corresponding constraint. This is important to make a redex explicit (*e.g.* in $x\ a\ [x = a \twoheadrightarrow b]$) or or to solve a match equation (*e.g.* in $a\ [a \ll x, x = a]$). As already mentioned, the substitution rule allows one to make the copies only upwards *w.r.t.* the order defined on the variables of ρ_{g}-terms. In the cyclic λ-calculus this is needed for the confluence of the system (see [2] for a counterexample) and it will be one of the key ingredients also for the confluence of the ρ_{g}-calculus. The *garbage* rules get rid of recursion equations that represent non-connected parts of the term. Matching constraints are not eliminated, keeping thus the trace of matching failures during an unsuccessful reduction. The *black hole* rules replace the undefined ρ_{g}-terms, intuitively corresponding to self-loop graphs, with the constant $\bullet$.

Note that all the evaluation steps are performed modulo the underlying theory associated to the "$\_, \_$" operator, as we will detail in the next section.

Example 2. [A simple reduction]

$$
\begin{array}{ll}
 & (f(a,a) \twoheadrightarrow a)\ (f(y,y)\ [y = a]) \\
\mapsto_{\mathcal{P}} & a\ [f(a,a) \ll f(y,y)\ [y = a]]\ \mapsto_{p}\ a\ [f(a,a) \ll f(y,y), y = a] \\
\mapsto_{dk} & a\ [a \ll y, a \ll y, y = a]\ =\ a\ [a \ll y, y = a]\ \textit{(by idempotency)} \\
\mapsto_{ac} & a\ [a \ll a, y = a]\ \mapsto_{dk}\ a\ [y = a]\ \mapsto\!\!\!\!\to_{gc}\ a
\end{array}
$$

3 Confluence of the ρ_{g}-calculus

The confluence for higher-order systems dealing with non-linear matching is difficult to get since we usually obtain non-joinable critical pairs as shown in [14] in the setting of the λ-calculus. Klop's counterexample can be encoded in the ρ-calculus [19] to show that ρ-calculus is not confluent if no evaluation strategy is used. The counterexample is still valid when generalizing the ρ-calculus to the ρ_{g}-calculus. For this reason, we restrict in the following to a linear ρ_{g}-calculus.

Definition 1 (Linear ρ_{g}-calculus). *A pattern is called* linear *if it does not contain two occurrences of the same variable. We say that a constraint* $[P_1 \lll G_1, \ldots, P_n \lll G_n]$ *is* linear *if all patterns are linear and if* $\bigcap_{i=1}^{n} \mathcal{FV}(P_i) = \emptyset$.

The linear ρ_{g}-calculus *is the ρ_{g}-calculus where all the patterns in the left-hand side of abstractions and all constraints are linear.*

As mentioned before, the "$\_, \_$" operator is supposed to be associative, commutative and idempotent, with ϵ as a neutral element. However, in the linear ρ_{g}-calculus, idempotency is not needed since constraints of the form $x \ll G, x \ll G$ are not allowed (and cannot arise from reductions). Therefore, in the ρ_{g}-calculus, rewriting must be thought of as acting over equivalence classes of ρ_{g}-terms with respect to $\sim_{AC_\epsilon}$, the congruence relation generated by the associativity and commutativity axioms for the "$\_, \_$" operator, and the neutrality

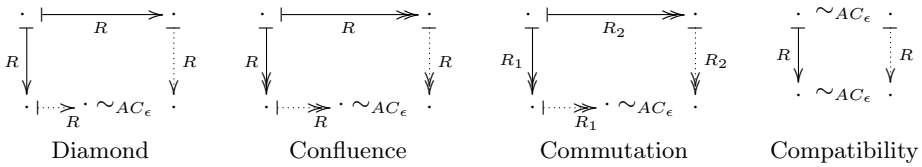


Fig. 3. Properties of rewriting modulo $\sim_E$

axiom for ϵ. If ρ_{g} denotes the rewrite system in Fig. 2, then the relation induced over AC_ϵ-equivalence classes is denoted $\mapsto_{\rho_{\mathsf{g}}/AC_\epsilon}$ and formally defined by $T_1 \mapsto_{\rho_{\mathsf{g}}/AC_\epsilon} T_2$ if $T_1 \sim_{AC_\epsilon} \mathsf{Ctx}\{\sigma(L)\}$ and $T_2 \sim_{AC_\epsilon} \mathsf{Ctx}\{\sigma(R)\}$ with $L \to R$ any rule in ρ_{g} and σ a substitution.

Concretely, in most of the proofs we will use the notion of rewriting modulo AC_ϵ à la Peterson and Stickel [17], denoted $\mapsto_{\rho_{\mathsf{g}},AC_\epsilon}$. In this case rewrite rules act on terms instead than on equivalence classes of terms, and matching modulo AC_ϵ is performed at each step of the reduction. Formally $T_1 \mapsto_{\rho_{\mathsf{g}},AC_\epsilon} T_2$ if $T_1 = \mathsf{Ctx}\{T\}$ with $T \sim_{AC_\epsilon} \sigma(L)$ and $T_2 = \mathsf{Ctx}\{\sigma(R)\}$. On one hand, this notion of rewriting is more convenient, from a computational point of view, than AC_ϵ-class rewriting. In fact in the latter case the entire (possibly infinite) class must be explored looking for reducible terms. On the other hand, as we will see later, to prove the confluence of the $\rho_{\mathsf{g}}, AC_\epsilon$ relation is sufficient to get the confluence for the $\rho_{\mathsf{g}}/AC_\epsilon$ relation.

Notation. In pictures and in the rest of the section, $\mapsto_R$ denotes the one step reduction and $\mapsto\!\!\!\!\!\rightarrow_R$ its reflexive and transitive closure, with R any subset of rules of the ρ_{g}-calculus. We often simply write AC_ϵ for $\sim_{AC_\epsilon}$ and $\mapsto_R$ for $\mapsto_{R,AC_\epsilon}$.

In Fig. 3 we give a graphical representation of some properties of the $\rho_{\mathsf{g}}, AC_\epsilon$ relation that will be referred to in the sequel. Their general definition can be found in [16]. The first three are ordinary properties from term rewriting, with the difference that the diagrams are closed with a step of equivalence modulo AC_ϵ. The last one says that if there exists a rewrite step from a term T, then the same step can be performed starting from any term AC_ϵ-equivalent to T.

The confluence proof, detailed in [6], is quite elaborated. This is mainly due to the non-termination of the system and to the fact that equivalence modulo AC_ϵ on terms has to be considered. We prove first a lemma showing the compatibility (see Fig. 3) of $\rho_{\mathsf{g}}, AC_\epsilon$ with AC_ϵ and thus ensuring that this relation is particularly well-behaved *w.r.t.* the congruence relation AC_ϵ. Then, we proceed by proving a number of lemmas that lead to the confluence of $\rho_{\mathsf{g}}, AC_\epsilon$ and finally we conclude on the confluence of $\rho_{\mathsf{g}}/AC_\epsilon$ using the mentioned compatibility lemma stated next.

Lemma 1 (Compatibility of $\rho_{\mathsf{g}}, AC_\epsilon$). *Compatibility with AC_ϵ holds for any rule in ρ_{g}.*

We point out that since compatibility holds for any rule of the ρ_{g}-calculus, then it also holds for any subset of evaluation rules of the ρ_{g}-calculus and, in particular, for the entire ρ_{g}-calculus semantics.

In order to prove the confluence of $\rho_{\mathsf{g}}, AC_\epsilon$, we proceed using a proof technique inspired from [2], but the larger number of evaluation rules of the ρ_{g}-calculus and the explicit treatment of the congruence relation on terms make the proof for the ρ_{g}-calculus more complex. The main idea is to split the rules into two subsets, to show separately their confluence and to prove the confluence of the union using a commutation lemma for the two sets of rules.

In the ρ_{g}-calculus the first subset, called the Σ-rules, consists of the two substitution rules *external sub* and *acyclic sub*, plus the δ rule. The second subset of rules, called the τ-rules, consists of all the remaining rules of the ρ_{g}-calculus. The substitution rules represent the non-terminating rules of the ρ_{g}-calculus. The δ rule is included in the Σ-rules, although it could be added to the τ-rules keeping this set of rules terminating. This choice is due to the fact that, because of its non-linearity, adding the δ rule to the τ-rules would have caused relevant problems in the proof of the final commutation lemma.

An important role in this part of the proof is played by the following proposition which follows immediately from [16].

Proposition 1. *1. A terminating relation locally confluent modulo AC_ϵ and compatible with AC_ϵ is confluent modulo AC_ϵ.*
2. The union of two relations commuting modulo AC_ϵ, both confluent modulo AC_ϵ and compatible with AC_ϵ, is confluent modulo AC_ϵ.

We start by showing the confluence modulo AC_ϵ of the τ-rules. This is done using Proposition 1.1). To prove the termination of the τ-rules classical but not trivial rewriting techniques are applied [12]: the lexicographic product of a polynomial interpretation on the ρ_{g}-terms and a path ordering induced by a given precedence on the operators of the ρ_{g}-calculus syntax is used. The local confluence modulo AC_ϵ is rather easy to prove by analysis of the critical pairs.

Proposition 2. *The relation τ is confluent modulo AC_ϵ.*

Secondly, we show the confluence modulo AC_ϵ of the Σ-rules and this is the most elaborated part of the proof. The difficulties arise from the fact that the rewrite relation induced by the Σ-rules is not strongly normalizing. Thus proving local confluence is not sufficient to get confluence. In particular, both substitution rules are not terminating in the presence of cycles:

$$x\ [x = f(y), y = g(y)] \to_{ac} x\ [x = f(g(y)), y = g(y)] \to_{ac} \dots$$
$$y\ [y = g(y)] \to_{es} g(y)\ [y = g(y)] \to_{es} \dots$$

The idea is to prove the confluence of the relation Σ by applying the complete development method of the λ-calculus, which consists first in defining a new rewrite relation *Cpl* with the same transitive closure as Σ and secondly in proving that the relation *Cpl* satisfies the diamond property modulo AC_ϵ (see

Fig. 3). Intuitively, a step of *Cpl* rewriting on a term G consists in the complete reduction of a set of redexes fixed initially in G. In other words, some redexes are marked in G and a complete development of these redexes is performed by the *Cpl* relation. Concretely, an underlining function is used to mark the redexes and reductions on underlined redexes are then performed using the following underlined version of the Σ-rules:

$$
\begin{array}{lll}
(\underline{\textit{external sub}}) & \mathsf{Ctx}\{\underline{y}\}\ [y = G, E] & \to_{\underline{es}}\ \mathsf{Ctx}\{G\}\ [y = G, E] \\
(\underline{\textit{acyclic sub}}) & G\ [G_0 \lll \mathsf{Ctx}\{\underline{y}\}, y = G_1, E] & \to_{\underline{ac}}\ G\ [G_0 \lll \mathsf{Ctx}\{G_1\}, y = G_1, E] \\
 & & \quad \textit{if } x > \underline{y},\ \forall x \in \mathcal{FV}(G_0) \\
(\underline{\delta}) & (G_1 \underline{;} G_2)\ G_3 & \to_{\underline{\delta}}\ G_1\ G_3 \underline{;} G_2\ G_3 \\
 & (G_1 \underline{;} G_2)\ [E]\ G_3 & \to_{\underline{\delta}}\ (G_1\ G_3 \underline{;} G_2\ G_3)\ [E]
\end{array}
$$

The term $x\ [x = f(\underline{y}), y = g(y)]$, for example, reach the $\underline{\Sigma}$ normal form $x\ [x = f(g(y)), y = g(y)]$ in one step. The *Cpl* rewrite relation is then defined as follows.

Definition 2. *Given the terms G_1 and G_2 in the Σ-calculus, we have that $G_1 \mapsto_{Cpl} G_2$ if there exists an underlining G_1' of G_1 such that $G_1' \mapsto\!\!\!\!\to_{\underline{\Sigma}} G_2$ and G_2 is in normal form w.r.t. the relation $\underline{\Sigma}$.*

For example, we have $G_1 = x\ [f(x, y) \ll f(\underline{z}, \underline{z}), z = g(\underline{w}), w = a] \mapsto_{\underline{\Sigma}} x\ [f(x, y) \ll f(\underline{z}, \underline{z}), z = g(a), w = a] \mapsto_{\underline{\Sigma}} x\ [f(x, y) \ll f(g(a), g(a)), z = g(a), w = a] = G_2$ and thus $G_1 \mapsto_{Cpl} G_2$.

To ensure that for every choice of redexes in G_1 there exists a *Cpl* reduction, we prove that $\underline{\Sigma}$ is weakly normalizing, by defining an appropriate reduction strategy. This property, in addition to the confluence property modulo AC_ϵ for $\underline{\Sigma}$ which can be proved using Proposition 1.1), allows us to prove the diamond property modulo AC_ϵ of the *Cpl* relation.

Lemma 2. *The relation Cpl satisfies the diamond property modulo AC_ϵ.*

Notice that if the relation *Cpl* has the diamond property modulo AC_ϵ, so does its transitive closure. The confluence of the Σ relation then follows easily by noticing that the Σ relation and the *Cpl* relation have the same transitive closure, that is $\mapsto_\Sigma \subseteq \mapsto_{Cpl} \subseteq \mapsto\!\!\!\!\to_\Sigma$. The first inclusion can be proved by underling the redex reduced by the Σ step. The second inclusion follows trivially from the definition of the *Cpl* relation.

Proposition 3. *The relation Σ is confluent modulo AC_ϵ.*

Finally, we consider the union of the subsets of rules τ and Σ. General confluence holds by the previous results and by the fact that we can prove the commutation of the τ-rules with the Σ-rules (see Proposition 1.2)).

Theorem 1 (Confluence of $\rho_{\mathsf{g}}, AC_\epsilon$). *The rewrite relation $\rho_{\mathsf{g}}, AC_\epsilon$ is confluent modulo AC_ϵ.*

As mentioned at the beginning, what we aim at is a more general result about rewriting on AC_ϵ-equivalence classes of ρ_{g}-terms. This can be achieved using the confluence of $\rho_{\mathsf{g}}, AC_\epsilon$ (Theorem 1) and the compatibility property of $\rho_{\mathsf{g}}, AC_\epsilon$ with AC_ϵ (Lemma 1).

Corollary 1 (Confluence of $\rho_{\mathsf{g}}/AC_\epsilon$). *The linear ρ_{g}-calculus is confluent.*

4 Term-graph rewriting in the ρ_{g}-calculus

The standard ρ-calculus can be seen as a natural generalization of both term rewriting and λ-calculus, integrating the pattern matching capabilities of the first formalism, with the abstraction mechanism of the second one. This fact has been formalized by showing that term rewriting can be simulated in the ρ-calculus [9,10].

The ρ_{g}-calculus has been already shown to be a quite expressive formalism which allows one to simulate both the plain ρ-calculus and the cyclic λ-calculus providing an homogeneous framework for pattern matching and higher-order graphical structures [7]. The possibility of representing structures with cycles and sharing naturally leads to the question asking whether first-order term-graph rewriting (TGR) can be simulated in this context. In this section we give a first positive answer. The complete proofs can be found in [6].

Several presentations have been proposed for TGR (see [18] for a survey). Here we consider an equational presentation in the style of [1], which is closer to the approach used in the ρ_{g}-calculus. Given a set of variables $\mathcal{X}$ and a first-order signature $\mathcal{F}$ with symbols of fixed arity, a term-graph over $\mathcal{X}$ and $\mathcal{F}$ is a system of equations of the form $G = \{x_1 \mid x_1 = t_1, \ldots, x_n = t_n\}$ where $t_1, \ldots, t_n$ are first-order terms over $\mathcal{X}$ and $\mathcal{F}$ and the recursion variables x_i are pairwise distinct. The variable x_1 on the left represents the root of the term-graph. We call the list of equations the *body* of the term-graph and we denote it by E_G, or simply E, when the graph G is clear from the context. The empty list is denoted by ϵ. The variables $x_1, \ldots, x_n$ are bound in the term-graph by the associated recursion equation. The other variables occurring in the term-graph G are called free and the set of free variables is denoted by $\mathcal{FV}(G)$. A term-graph without free variables is called closed. We denote the collection of variables appearing in G by $\mathcal{V}ar(G)$. Two α-equivalent graphs, *i.e.* two graphs which differ only for the name of bound variables, are considered equal. Cycles may appear in the system and degenerated cycles, *i.e.* equations of the form $x = x$, are replaced by $x = \bullet$ (black hole). A term-graph is said to be in *flat form* if all its recursion equations are of the form $x = f(x_1, \ldots, x_n)$, where the variables $x, x_1, \ldots, x_n$ are not necessarily distinct from each other. In the following we will consider only term-graphs in flat form and without useless equations (garbage) that we remove automatically during rewriting. A term-graph in flat form can be easily interpreted and depicted as a graph taking the set of variables as nodes. We will use the graphical interpretation to help the intuition in the examples.

Rewriting is done by means of term-graph rewrite rules. A *term-graph rewrite rule* is a pair of term-graphs (L, R) such that L and R have the same root, L

is not a single variable and $\mathcal{FV}(R) \subseteq \mathcal{FV}(L)$. We say that a rewrite rule is *left-linear* if L is a tree. In the sequel we will restrict to left-linear rewrite rules.

Definition 3 (Variable substitution). *A substitution $\sigma = \{x_1/y_1, \ldots, x_n/y_n\}$ is a map from variables to variables. Its application to a term-graph G, denoted $\sigma(G)$ is defined as follows:*

$$\sigma(x) \triangleq \begin{cases} y_i & \text{if } x = x_i \in \{x_1, \ldots, x_n\} \\ x & \text{otherwise} \end{cases} \qquad \sigma(f(G_1, \ldots, G_n)) \triangleq f(\sigma(G_1), \ldots, \sigma(G_n))$$

$$\sigma(\{x_1 \mid x_1 = G_1, \ldots, x_n = G_n\}) \triangleq \{\sigma(x_1) \mid \sigma(x_1) = \sigma(G_1), \ldots, \sigma(x_n) = \sigma(G_n)\}$$

A rewrite rule can be applied to a term-graph G if there exists a match between its left-hand side and the graph. Formally, a *homomorphism (matching)* from a term-graph L to a term-graph G is a substitution σ such that $\sigma(L) \subseteq G$ where the inclusion means that all the recursion equations in $\sigma(L)$ are present also in G.

A *redex* in a term-graph G is a pair $((L, R), \sigma)$ where (L, R) is a rule and σ is an homomorphism from the left-hand side L of the rule to G. If x is the root of L, we call $\sigma(x)$ the head of the redex.

Definition 4 (Path, position). *A* path *in a closed graph G is a sequence of function symbols interleaved by integers $p = f_1 i_1 f_2 \ldots i_{n-1} f_n$ such that f_{j+1} is the i_j-th argument of f_j, for all $j = 0 \ldots n$. The sequence of integers $i_1 \ldots i_{n-1}$ is called the* position *of the node labeled f_n and still denoted with the letter p. By the context notation $G_{\lceil E_{G'} \rceil_p}$ we specify that G contains the body of a graph G' at the position p.*

The notions of path and position are used to define a rewrite step. Let $((L, R), \sigma)$ be a redex occurring in G at the position p. A *rewrite step* consists in removing the equation specified by the head of the redex and in replacing it by the body of $\sigma(R)$, with a fresh choice of bound variables. Using a context notation we write $G_{\lceil \sigma(x)=t \rceil_p} \to G_{\lceil \sigma(E_R) \rceil_p}$.

The set of term-graphs of a TGR is a strict subset of the set of terms of the ρ_{g}-calculus, modulo some obvious syntactic conventions. By abuse of notation, in the following we will consider equivalent the two notations $\{x \mid E\}$ and $x\ [E]$. A rewrite rule $(L_i, R_i) \in \mathcal{R}$ is translated into the corresponding ρ_{g}-term $L_i \twoheadrightarrow R_i$. The application of a substitution $\sigma = \{x_1/G_1, \ldots x_n/G_n\}$ to a term-graph L corresponds in the ρ_{g}-calculus to the addition of a list of constraints to the term L, that is $L\ [E]$ where $E = (x_1 = G_1, \ldots, x_n = G_n)$.

A ρ_{g}-term is, in general, more complex than a flat term-graph, *i.e.* it can have garbage and nested lists of constraints. We define next the canonical form of a ρ_{g}-term G containing no abstractions and no match equations.

Definition 5 (Canonical form). *Let G be a ρ_{g}-term containing no abstractions and no match equations. We say that G is in canonical form, denoted $\overline{G}$, if it is in flat form and it contains neither garbage equations nor trivial equations of the form $x = y$.*

To reach the canonical form, we first perform the flattening and merging of the lists of equations of G and we introduce new recursion equations with fresh variables for every subterm of G. We obtain in this way a ρ_{g}-term in flat form, where the notion of flat form is similar to the one defined for term-graphs. The canonical form can then be obtained from the flat form by removing the useless equations, by means of the two substitution rules and the garbage collection rule of the ρ_{g}-calculus. We point out that the canonical form of a ρ_{g}-term is unique, and a ρ_{g}-term in canonical form corresponds to a term-graph in flat form. Before proving the correspondence of rewritings, we need a lemma showing that matching in the ρ_{g}-calculus is well-behaved *w.r.t.* the notion of homomorphism.

Lemma 3 (Matching). *Let G be a closed term-graph and let (L, R) be a rewrite rule, with $\mathcal{V}ar(L) = \{x_1, \ldots, x_m\}$, such that L is homomorphic to G using the variable renaming $\sigma = \{x_1/x'_1, \ldots, x_m/x'_m\}$.*
Let $E = (x_1 = x'_1, \ldots, x_n = x'_n, E_G)$ with $\{x_1, \ldots, x_n\} = \mathcal{FV}(L)$. Then in the ρ_{g}-calculus we have $L \ll G \mapsto\!\!\!\!\rightarrow_{\rho g} E$ and $\overline{L\ [E]}$ is homomorphic to G.

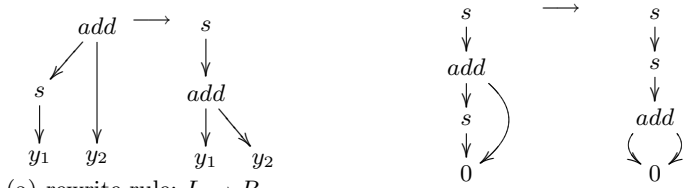
This result guarantees the fact that if there exists an homomorphism, *i.e.* a variable renaming, between two term-graphs, in the ρ_{g}-calculus we obtain the variable renaming (in the form of a list of recursion equations) as result of the evaluation of the matching problem generated from the two graphs. In other words, this means that if a rewrite rule can be applied to a term-graph, the application is still possible when passing to the ρ_{g}-calculus side.

Example 3 (Matching). Consider the two term-graphs $L = \{x_1 \mid x_1 = add(x_2, y_1),\ x_2 = s(y_2)\}$ and $G = \{z_0 \mid z_0 = add(z_1, z_2),\ z_1 = s(z_2), z_2 = 0\}$ (see Figure 4). Then the substitution $\sigma = \{x_1/z_0,\ x_2/z_1,\ y_1/z_2,\ y_2/z_2\}$ is an homomorphism from L to G. We show how the substitution can be obtained in the ρ_{g}-calculus starting from the matching problem $L \ll G$. We use the notation $\mapsto\!\!\!\!\rightarrow_{r,s}$ to express two steps $\mapsto_r \mapsto_s$, where r and s are two ρ_{g}-rules.

$$
\begin{array}{lll}
L \ll G \mapsto_{p} & L \ll z_0, E_G & \\
\quad \mapsto\!\!\!\!\rightarrow_{es,gc} & add(s(y_2), y_1) \ll z_0, E_G & \\
\quad = & add(s(y_2), y_1) \ll z_0, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 & \\
\quad \mapsto_{ac} & add(s(y_2), y_1) \ll add(z_1, z_2), z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 & \\
\quad \mapsto_{dk} & s(y_2) \ll z_1, y_1 \ll z_2, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 & \\
\quad \mapsto\!\!\!\!\rightarrow_{ac,dk} & y_2 \ll z_2, y_1 \ll z_2, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 & \\
\quad \mapsto\!\!\!\!\rightarrow_{s} & y_2 = z_2, y_1 = z_2, E_G &
\end{array}
$$

We can verify then that $\overline{L\ [y_2 = z_2, y_1 = z_2, E_G]}$ is homomorphic to G. In fact, the transformation into the canonical form leads to the graph $x_1\ [x_1 = add(x_2, z_2), x_2 = s(z_2), z_2 = 0]$ and it is easy to see that the substitution $\tau = \{x_1/z_0, x_2/z_1\}$ makes this graph homomorphic to G.

We next prove that any term-graph rewrite step can be simulated in the ρ_{g}-calculus. Since in the ρ_{g}-calculus the rule application is at the object-level, we need to define a ρ_{g}-term encoding the position of the redex in the given term-graph.



(a) rewrite rule: $L \to R$ (b) rewrite step: $G \to G'$

Fig. 4. Example of rewriting

Definition 6 (Position trace graph). *Let $p = f\ i\ p'$ be a path in a graph G and $x_0, \ldots, x_j, \ldots$ be a set of fresh dinstinct variables. The* position trace graph *$P_p(G)$ is recursively defined as $P_\epsilon(G) = x_0$ and $P_p(G) = f(x_1, \ldots, P_{p'}(G), \ldots, x_n)$ where f is of arity n and has $P_{p'}(G)$ as i-th argument. We assume that every variable x_j is used only once in the construction of $P_p(G)$.*

The position trace graph is then used to build a ρ_{g}-term H that pushes the rewrite rule down to the correct application position, according to the given term-graph rewrite step.

Lemma 4 (Simulation). *Given a term-graph G and a rewrite rule (L, R) such that $G_{\lceil \sigma(z)=t \rceil_p} \to G_{\lceil \sigma(E_R) \rceil_p} = G'$. Let $H \triangleq y \twoheadrightarrow (P_p(G)_{\lceil x \rceil_p} \twoheadrightarrow P_p(G)_{\lceil y\ x \rceil_p})$, then in the ρ_{g}-calculus we have the reduction $(H\ (L \twoheadrightarrow R)\ G) \mapsto\!\!\!\!\!\rightarrow_{\rho g} G''$ with $\overline{G''}$ homomorphic to G'.*

The final ρ_{g}-term we obtain is not exactly the same as the term-graph resulting from the ρ_{g}-reduction in the TGR, and this is due to some unsharing steps that may occur in the reduction. In general, we have an homomorphism between the two graphs and this corresponds to the fact that, in presence of cycles, the ρ_{g}-term is possibly more "unraveled" than the term-graph G'.

Example 4 (Addition). Let (L, R), where $L = \{x_1 \mid x_1 = add(x_2, y_1),\ x_2 = s(y_2)\}$ and $R = \{x_1 \mid x_1 = s(x_2),\ x_2 = add(y_1, y_2)\}$, be a rewrite rule describing the addition of natural numbers. We apply this rule to the term-graph $G = \{z \mid z = s(z_0), z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0\}$ using the variable renaming $\sigma = \{x_1/z_0, x_2/z_1, y_1/z_2, y_2/z_2\}$. We obtain $G' = \{z \mid z = s(z_0), z_0 = s(z_1'), z_1' = add(z_2, z_2), z_2 = 0\}$. For a graphical representation see Fig. 4.

The corresponding reduction in the ρ_{g}-calculus is as follows. First of all, since the rule is not applied at the head position of G, we need to define the ρ_{g}-term $H = y \twoheadrightarrow s(x) \twoheadrightarrow s(y\ x)$ that pushes down the rewrite rule to the right application position, *i.e.* under the symbol s. We obtain the reduction

$$\begin{aligned} &(y \twoheadrightarrow s(x) \twoheadrightarrow s(y\ x))\ (L \twoheadrightarrow R)\ G \\ \mapsto\!\!\!\!\!\rightarrow_{\rho g}\ &s(x) \twoheadrightarrow s((L \twoheadrightarrow R)\ x)\ \ G \end{aligned}$$

$\mapsto_\rho\ s((L \multimap R)\ x)\ [s(x) \ll G]$
$\mapsto_p\ s((L \multimap R)\ x)\ [s(x) \ll z, E_G]$
$\mapsto\!\!\!\!\rightarrow_{pg}\ s((L \multimap R)\ z_0)\ [E_G]$
$\mapsto_\rho\ s(R\ [L \ll z_0])\ [E_G]$
$\mapsto\!\!\!\!\rightarrow_{pg}\ s(R\ [y_1 = z_2, y_2 = z_2])\ [E_G]$
$=\quad s(x_1\ [x_1 = s(x_2), x_2 = add(y_1, y_2)]\ [y_1 = z_2, y_2 = z_2])\ [E_G]$
$\mapsto\!\!\!\!\rightarrow_{pg}\ s(x_1\ [x_1 = s(x_2), x_2 = add(z_1, z_2)])\ [E_G] = G''$

The canonical form of G'' is then obtained by removing the useless recursion equations in E_G and merging the lists of constraints. We get $\overline{G''} = x\ [x = s(x_1), x_1 = s(x_2), x_2 = add(z_1, z_2), z_0 = 0]$. The graph $\overline{G''}$ is homomorphic (in this case even equal up to variable renaming) to the term-graph G'.

5 Conclusions

The ρ_{g}-calculus is an extension of the ρ-calculus that allows one to represent and compute over regular infinite entities. It represents a common framework where higher-order capabilities, graphical structures and matching are primitive features, leading to a quite expressive calculus. The ρ_{g}-calculus terms are grouped into equivalence classes defined according to the theory specified for the constraint conjunction operator, which in general is the associative, commutative, idempotent theory with neutral element ϵ. If we restrict to a linear ρ_{g}-calculus, since all constraints are linear and this property is obviously preserved by reduction, we do not need to work modulo the idempotency axiom. We have shown here that, choosing this underlying theory for the conjunction operator for constraints, the linear ρ_{g}-calculus enjoys the confluence property on equivalence classes of terms.

In [7] the ρ_{g}-calculus has been shown to be an expressive formalism able to simulate both the standard ρ-calculus and the cyclic λ-calculus. We have shown in this paper that also term-graph rewrite systems can be encoded in the ρ_{g}-calculus. More precisely we have shown that for every rewriting step in a TGR we can build a ρ_{g}-term which simulates such rewriting as a sequence of reductions in the ρ_{g}-calculus. We have not investigated here the conservativity issue, but we believe that a positive result can be obtained exploiting the confluence property of the ρ_{g}-calculus. The main difference between the two systems lies in the fact that rewrite rules and their control (application position) are defined at the object-level of the ρ_{g}-calculus while in the TGR the reduction strategy is left implicit. The possibility of controlling the application of rewrite rules is particularly useful when the rewrite system is not terminating. It would be certainly interesting to define in the ρ_{g}-calculus iteration strategies and strategies for the generic traversal of terms in order to simulate TGR rewritings guided by a given reduction strategy. A similar work has already been done for representing first-order term rewriting reductions in a typed version of the ρ-calculus [10]. Intuitively, the ρ-term encoding a first-order rewrite systems is a ρ-structure consisting of the corresponding term rewrite rules wrapped in an iterator that allows for the repetitive application of the rules. We conjecture that this approach can

be adapted and generalized for handling term-graphs and simulate term-graphs reductions.

References

1. Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3-4):207–240, 1996.
2. Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154–233, 1997.
3. H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier. 1984. Second edition.
4. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In *Proc. of PARLE'87*, vol. 259 of *LNCS*, pp. 141–158, Eindhoven, 1987. Springer-Verlag.
5. G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In *Proc. of POPL'03*, pp. 250–261. ACM Press, 2003.
6. C. Bertolissi. The graph rewriting calculus: proof of confluence and simulation of TGRs. Technical report, INRIA-LORIA, 2005.
7. C. Bertolissi, P. Baldan, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. In *Proc. of TERMGRAPH'04*, vol. 127(5) of *ENTCS*, pp. 21–41, 2005.
8. H. Cirstea, G. Faure, and C. Kirchner. A ρ-calculus of explicit constraint application. In *Proc. of WRLA'04*, vol. 117 of *ENTCS*, pp. 51–67, 2004.
9. H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, 2001.
10. H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *Proc. of TYPES'03*, vol. 3085 of *LNCS*, pp. 147–171, 2003.
11. A. Corradini. Term rewriting in CT_Σ. In *Proc. of TAPSOFT'93*, vol. 668 of *LNCS*, pp. 468–484, 1993.
12. N. Dershowitz. Termination of rewriting. *Journal Symbolic Computation*, 3(1-2):69–116, 1987.
13. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. In *Proc. of POPL'84*, pp. 83–92. ACM Press, 1984.
14. J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, 1980.
15. L. Liquori and B. Serpette. iRho: an Imperative Rewriting Calculus. In *Proc. of PPDP'04*, pp. 167–178. ACM Press, 2004.
16. E. Ohlebusch. Church-Rosser theorems for abstract reduction modulo an equivalence relation. In *Proc. of RTA'98*, vol. 1379 of *LNCS*, pp. 17–31. Springer, 1998.
17. G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28:233–264, 1981.
18. M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, eds. *Term graph rewriting: theory and practice*. Wiley, London, 1993.
19. B. Wack. Klop counter example in the ρ-calculus. Draft notes, LORIA, Nancy, 2003.